FACULTAT D'INFORMÀTICA DE BARCELONA

BACHELOR'S DEGREE IN INFORMATICS ENGINEERING

Credit Score Classification

DATA MINING COURSE

Members:

Pol Pérez Castillo

Maxime Côté-Lapointe

Daniel Muñoz Arroyo

Alejandro Salvat Navarro

Index

Motivation

The use of credit scores has become increasingly important in the financial industry, as they are widely used to assess creditworthiness and determine the interest rates and loan terms that individuals and businesses are eligible for. Developing accurate and efficient credit scoring models can save financial institutions significant amounts of time and resources, while also providing customers with fair and transparent lending decisions. In this project, we aim to leverage data mining techniques to build a credit scoring model for a global finance company, using a dataset that contains basic bank details and various credit-related features.

Removal of useless features

So first, before doing cleaning and preprocessing, right off the bat, we can usually remove features that are useless and anonymize our dataset by removing identifying features such as:

'ID', 'Customer_ID', 'Name' and 'SSN' are all identifiers of some sort and won't help us predict the credit scores of people, hence why they were dropped.

'Month' shouldn't really have an effect as well even though people might spend differently near Christmas or other holidays, but we'll assume in general that their habits should be visible across the whole year. Example: if they have bad spending habits, it won't only happen at some specific time of the year.

As for 'Occupation', it isn't clear if 'Annual_Income' can give as much information as it so it will be kept for now. We know that occupations usually correlate to annual income, so we'll keep that in mind going forward.

Metadata

The dataset we are working with contains information on 150,000 individuals and includes 28 columns (before preprocessing) providing a range of credit and finance-related information. The data covers basic bank details such as account balances, credit limits, and the number of bank accounts and credit cards held. It also includes more complex credit-related information such as the number and types of loans taken out, payment behavior, outstanding debts, credit history, and credit utilization ratio. Additionally, the dataset includes columns related to income and expenses such as monthly in-hand salary, monthly balance, and total EMI per month. Understanding the significance and impact of each of these columns will be crucial in developing an accurate credit scoring model.

Now, one by one, we are going to explain all the important features we have in our data set:

- **Age**
  This column represents the age of the person.

- **Occupation**
  This feature represents the occupation of the person.

- **Annual Income**
  This variable represents the annual income of the person.

- **Monthly Inhand Salary**
  This column represents the monthly base salary of a person.

- **Num Bank Accounts**
  This feature represents the number of bank accounts a person holds.

- **Num Credit Card**
  This variable represents the number of other credit cards held by a person.

- **Interest Rate**
  This column represents the interest rate on credit cards.

- **Num of Loan**
  This feature represents the number of loans taken from the bank.

- **Delay from due date**
  This variable represents the average number of days delayed from the payment date.

- **Num of Delayed Payment**
  This column represents the average number of payments delayed by a person.

- **Changed Credit Limit**
  This feature represents the percentage change in credit card limit.

- **Num Credit Inquiries**
  This variable represents the number of credit card inquiries.

- **Credit Mix**
  This column represents the classification of the mix of credits.

- **Outstanding Debt**
  This feature represents the remaining debt to be paid (in USD)

- **Credit Utilization Ratio**
  This variable represents the utilization ratio of credit cards.

- **Credit History Age**
  This column represents the age of credit history of the person.

- **Payment of Min Amount**
  This feature represents whether only the minimum amount was paid by the person.

- **Total EMI per month**
  This variable represents the monthly EMI (equated monthly installments) payments (in USD).

- **Amount invested monthly**
  This column represents the monthly amount invested by the customer (in USD).

- **Payment Behaviour**
  This feature represents the payment behavior of the customer (in USD).

- **Monthly Balance**
  This variable represents the monthly balance amount of the customer (in USD).

- **Credit Score**
  The response variable represents the bracket of credit score (Poor, Standard, Good)

Preprocessing

Treatment of the data

When we opened the test.csv, we saw that there were blank spaces in the predicted variable, so we can't train the model if we don't have the objective variable. The only data that we are working with it is the train.csv. This data have the objective variable, but if we have a look in the whole original dataset we can see values that are in an incorrect format to be treated, because if we import the dataset, the majority of instances are strings and have strange values like '__1000__', blank spaces or values like '!@9#%8'. In order to correct that, we erase the '_' and '!@9#%8' characters by replacing to the black character .

Outliers

When studying credit cards or any other dataset, it is important to address outliers during the preprocessing stage.

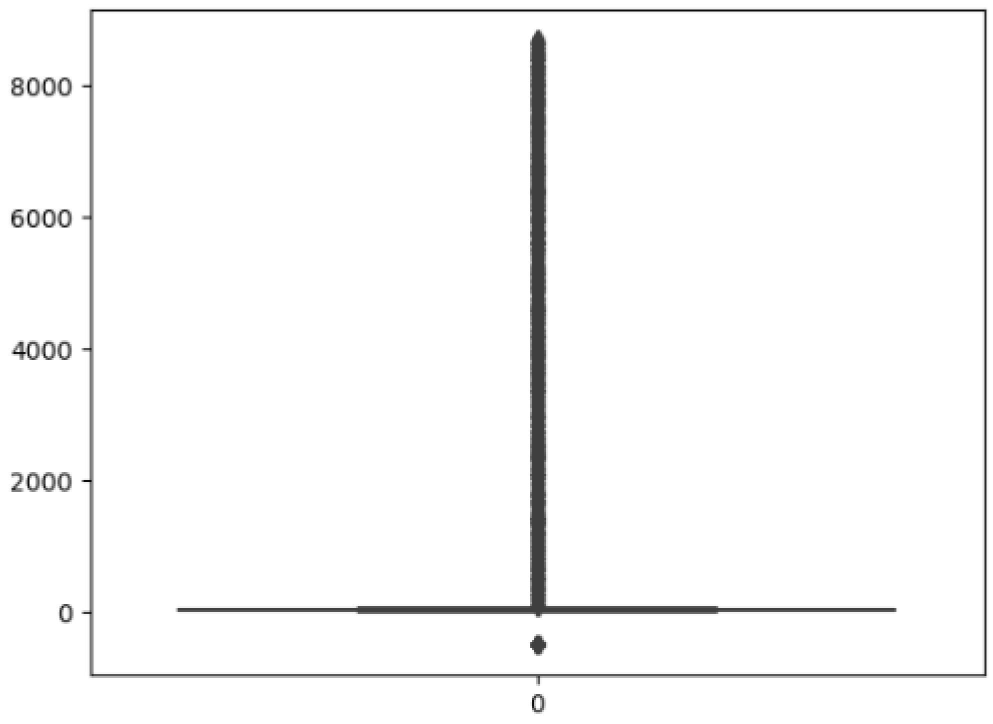
The first step was to identify the outliers in our dataset. This was done through visual inspection using techniques such as box plots. Then we understood the context of the data and the reasons behind their occurrence, because it's possible that some outliers are legitimate data points representing important and meaningful information. And finally,for the treatment of those outliers we decided to impute them with a more reasonable value based on the distribution of the data.

Example:

- Variable 'Age' with strange outliers

```
sns.boxplot(df['Age'])
```

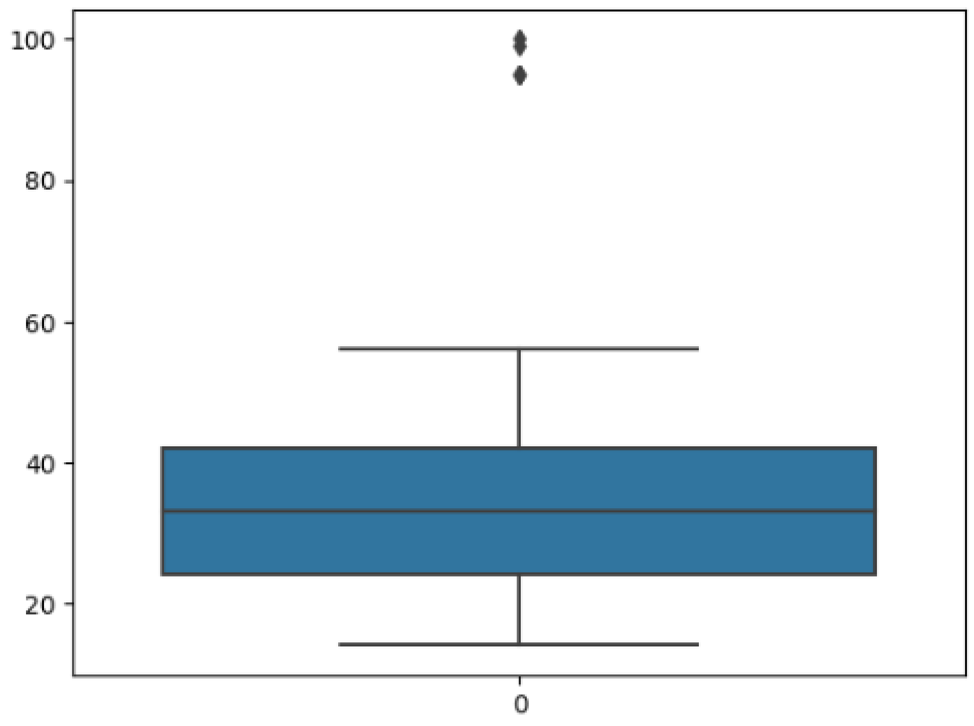
```
<Axes: >
```



- Variable ‘Age’ with understandable data

```
sns.boxplot(df['Age'])
```

```
<Axes: >
```

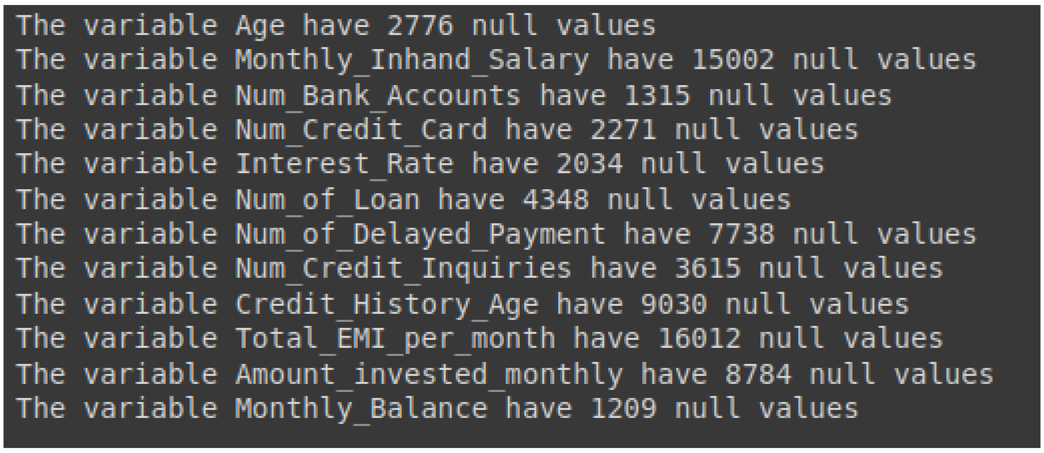


Treatment of the variable with null values

In this section, we change the format of the variable that isn’t in the correct form. This is the case of Credit_History_Age variable, it is in the next format: ‘ny years and nm months’ where ny are the number of years and nm the number of months, so we transform to numerical variable by transforming to a float (the whole part is the number of ages and the decimal part are the number of months/12).
For the rest of numerical variables with null values, we extract from the string the numbers and we use it.

Treatment of null values

As we can see in the next picture, there are variables with null values:

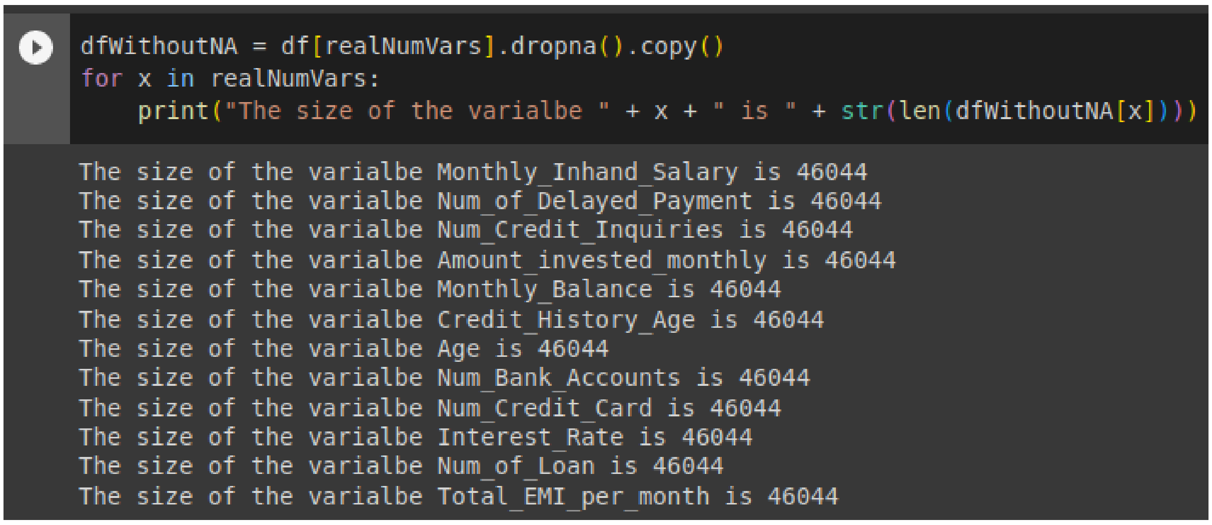

```
The variable Age have 2776 null values
The variable Monthly_Inhand_Salary have 15002 null values
The variable Num_Bank_Accounts have 1315 null values
The variable Num_Credit_Card have 2271 null values
The variable Interest_Rate have 2034 null values
The variable Num_of_Loan have 4348 null values
The variable Num_of_Delayed_Payment have 7738 null values
The variable Num_Credit_Inquiries have 3615 null values
The variable Credit_History_Age have 9030 null values
The variable Total_EMI_per_month have 16012 null values
The variable Amount_invested_monthly have 8784 null values
The variable Monthly_Balance have 1209 null values
```

We can not keep the null values, because in the project we will use models that don't accept null values in the data, so keeping null values is not an option.

All of those variables are numerical and we are going to try 3 different methods to treat the null values and we will use one of them and then use one of them to execute the models. Those methods are the next: dropping NA, impute the values with the mean and use KNN imputer to try to impute a value comparing with similar individuals and try to see if the distribution doesn’t change a lot or in the case of drop the null variables if we are losing a lot of information or not.

Dropping Null values

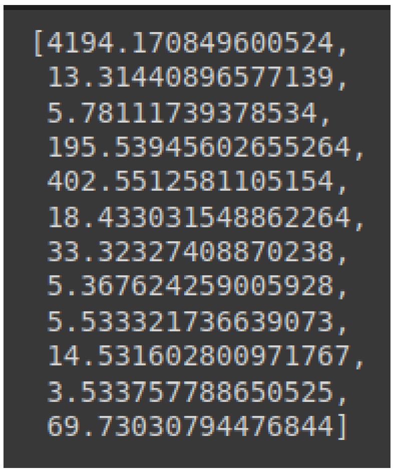
With this treatment, we are going to try to drop the null values and see what it happens:


```python
dfWithoutNA = df[realNumVars].dropna().copy()
for x in realNumVars:
    print("The size of the varialbe " + x + " is " + str(len(dfWithoutNA[x])))
```
```
The size of the varialbe Monthly_Inhand_Salary is 46044
The size of the varialbe Num_of_Delayed_Payment is 46044
The size of the varialbe Num_Credit_Inquiries is 46044
The size of the varialbe Amount_invested_monthly is 46044
The size of the varialbe Monthly_Balance is 46044
The size of the varialbe Credit_History_Age is 46044
The size of the varialbe Age is 46044
The size of the varialbe Num_Bank_Accounts is 46044
The size of the varialbe Num_Credit_Card is 46044
The size of the varialbe Interest_Rate is 46044
The size of the varialbe Num_of_Loan is 46044
The size of the varialbe Total_EMI_per_month is 46044
```
As we can see in the above picture, if we drop all the rows with null values, we will pass from 100000 instances to 46044 and that implies that we are losing 54 % of the information in the data.

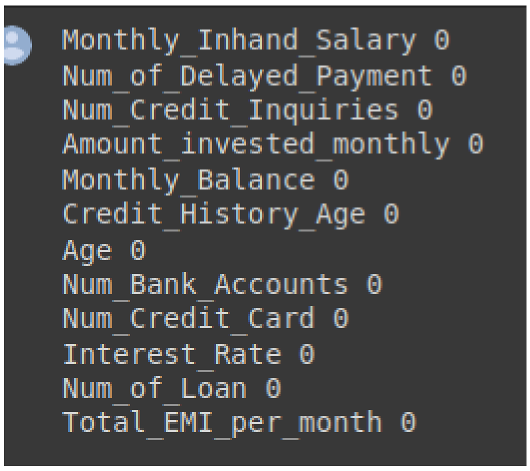
We think that is a lot of information and for that reason is not a good idea to work with it, so, we are going to see other techniques in order to impute the null values.

Mean imputation

We are trying to replace the nan values in each variable by his mean.
First of all, we calculated the mean, as we can see in the bottom left picture, there are the mean and on the bottom-right picture, we can see that now, any treated variable have null values, so we replaced it well.



```
[4194.170849600524,
 13.31440896577139,
 5.78111739378534,
 195.53945602655264,
 402.5512581105154,
 18.433031548862264,
 33.32327408870238,
 5.367624259005928,
 5.533321736639073,
 14.531602800971767,
 3.533757788650525,
 69.73030794476844]
```

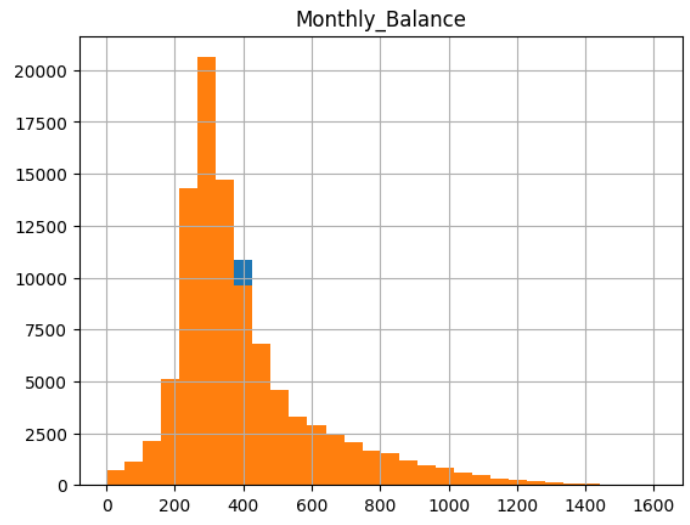


```
Monthly_Inhand_Salary 0
Num_of_Delayed_Payment 0
Num_Credit_Inquiries 0
Amount_invested_monthly 0
Monthly_Balance 0
Credit_History_Age 0
Age 0
Num_Bank_Accounts 0
Num_Credit_Card 0
Interest_Rate 0
Num_of_Loan 0
Total_EMI_per_month 0
```

After that, we can see how it affects the distribution of the variables by plotting each one using a histogram with the mean imputation (blue color) and then override it with the variable in the originalDataset without null values (orange color). Finally, we will see the original distribution without null values in orange and the blue var will be the extra individuals that we replace with the mean.

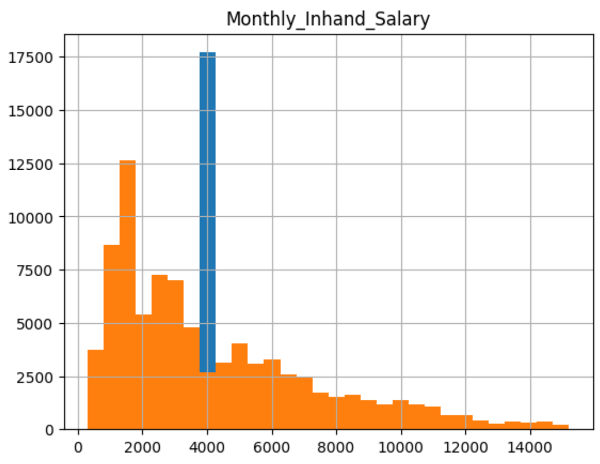
In the code we can see all the histograms for all the variables with null values, but I am going to show only some of the plots that we consider that are relevant.

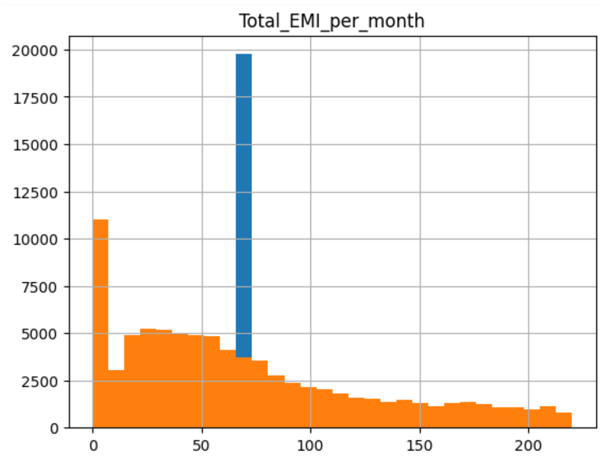
Something interesting that we can see is that if the variables that have a small quantity of null values, the distribution doesn't change a lot. It happens for example in Monthly_Balance::



This variable, as we can see in the picture above, if we compare the orange distribution with the blue plus orange distribution, it doesn't change a lot. And if we have a look at the first picture in NA imputation, we can see that it has only 1209 null values, that is the variable with the lowest null values greater than 0.

But if there are more null values, the distribution changes like in the next variables:





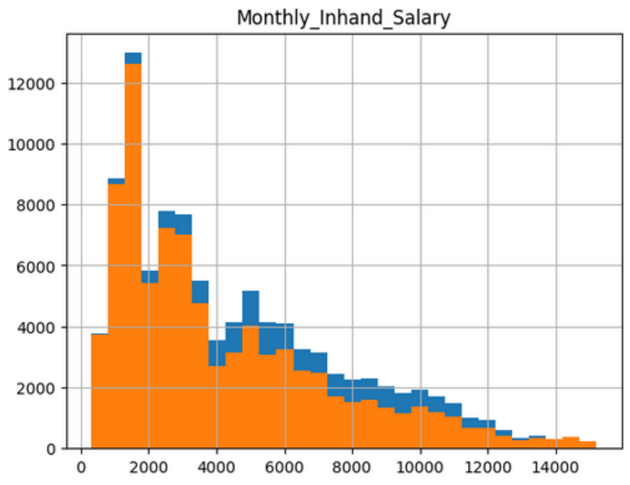
As we can see, initially (in orange) the distribution where more or less exponential, but then it changes a lot by the mean imputation (as we can see in the blue + orange distribution) because they have 15002 and 16012 values respectively, and all of those values are the same (the mean)

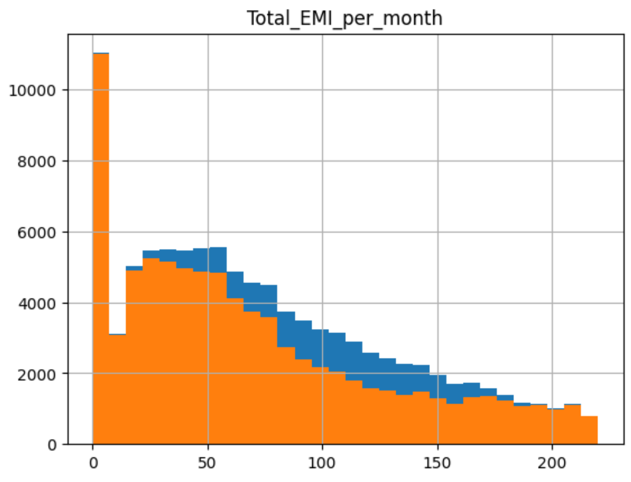
We think that this is not a good option because in the cases with higher variables, it doesn’t happen, so we are going to see another method in order to correct that.

KNN imputation

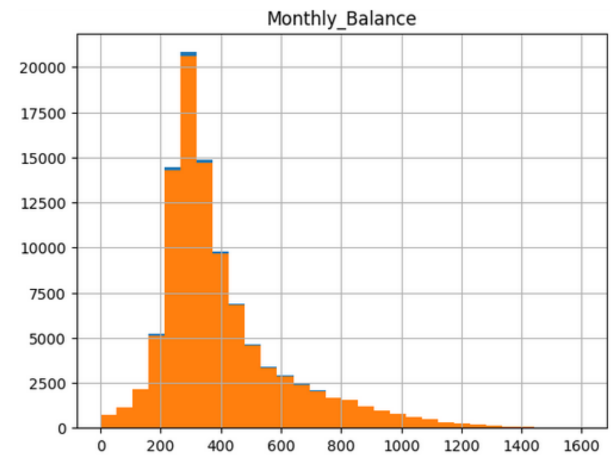
Finally, in order to correct the problem in the mean imputation, we are going to do the imputation of null values working with the KNN imputer because it impute the values comparing with the K nearest neighbors (in this case K = 5) and it shouldn’t assign all the null values in the same variable with the same value, so the distribution will be different than in the mean imputation.

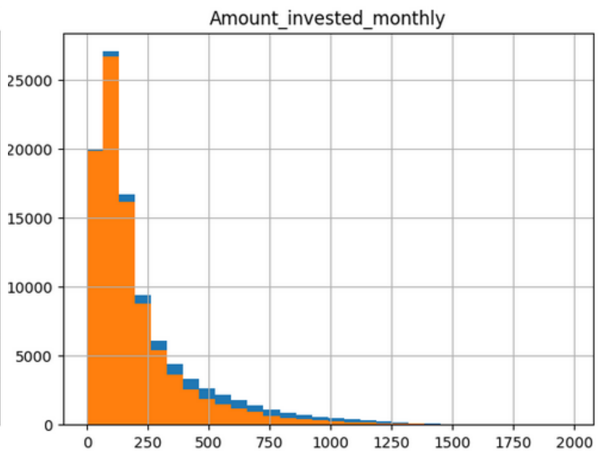
If we have a look in the two variables that make problem in the mean imputation:

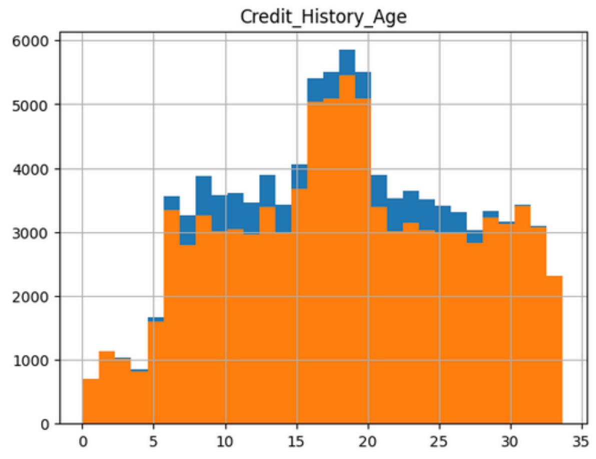


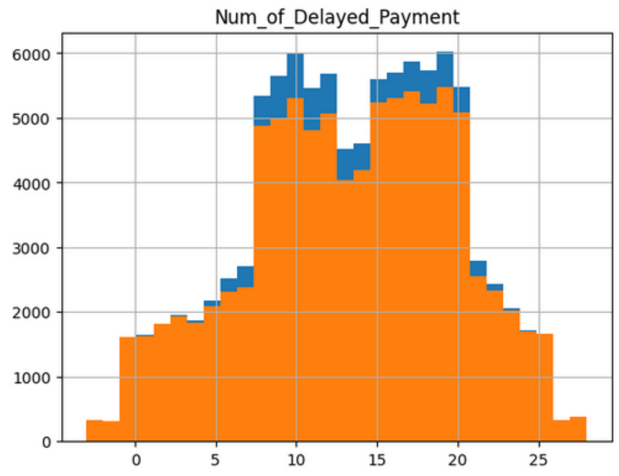


As we can see in the two pictures above, the distribution is more or less the same.
And if we look on the other variables:









The first two variables have small numbers of null values and it follows better the initial distribution, and the other two, are the 3th and 4th higher number of null values, but it continues following more or less the same distribution.

Comparing the three methods, in order to don’t lose the 54 % information the better imputation method between the mean and KNN imputation, is the second one, because it follows better the original distribution with low numbers of null values, but it continue following more or less the same distribution than the original with the higher values

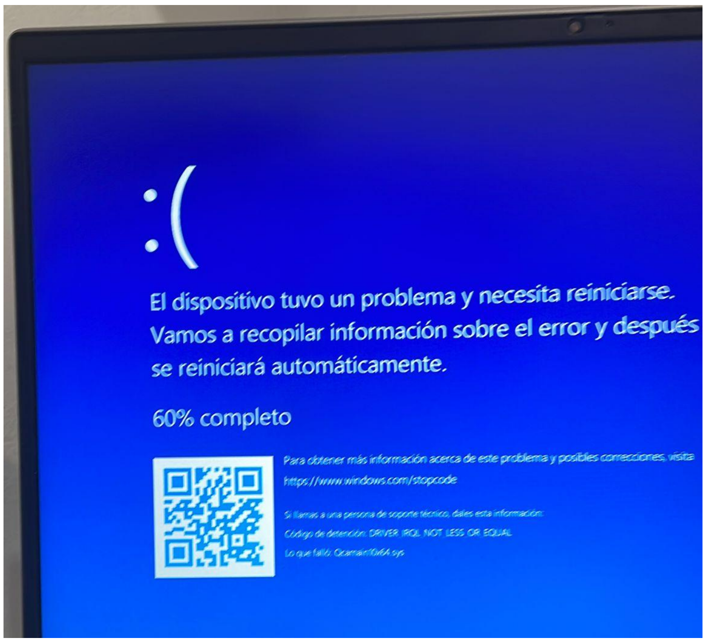
One hot encoding

To preprocess the categorical variables in our dataset, we first decided to use the one hot coding technique, as these categorical variables have no numerical order or relationship between categories, so assigning numerical indices to them might provide slightly worse results. In addition, one hot coding allows us to identify and understand the individual impact of each category on the model's outcome. The categorical variables to which we applied this pre-processing were: Occupation and Payment behaviour. The problem is that, by using this technique, we doubled the number of total columns, greatly increasing the execution time of our code and making the project very difficult to implement. So, finally, we decided to assign numeric indices to our categorical variables

Problem executing the data

Ideally, the KNN imputer was the best option in order to maintain all the information about the data and treat the null values to work with all the methods, including those that doesn't accept the null values.

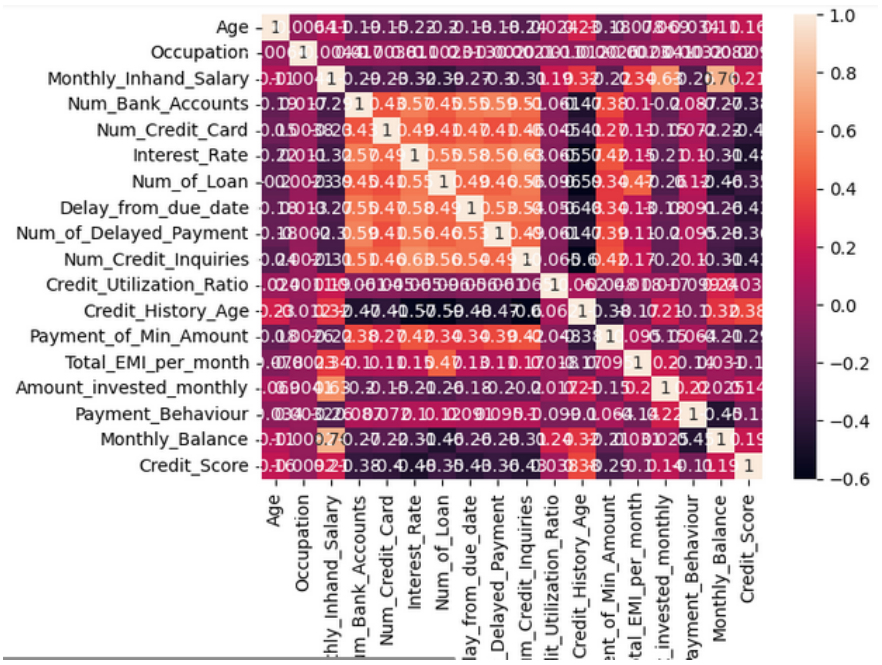
When we export the Cleaned_Data, we couldn't execute the models because we were waiting about 50 minutes and it didn't finish, and also, the following error was thrown while we were running one of the models:



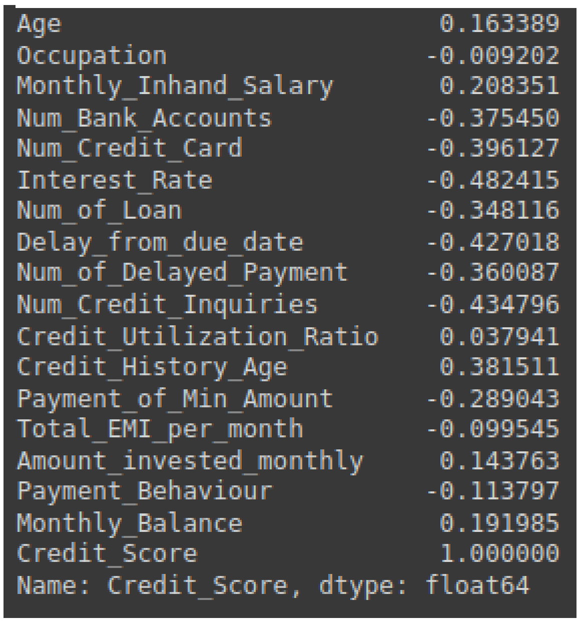
```
:(

El dispositivo tuvo un problema y necesita reiniciarse.
Vamos a recopilar información sobre el error y después
se reiniciará automáticamente.

60% completo

Para obtener más información acerca de este problema y posibles correcciones, visita
https://www.windows.com/stopcode

Si llamas a una persona de soporte técnico, dales esta información:
Código de detención: DRIVER_IRQL_NOT_LESS_OR_EQUAL
Lo que falló: Qcamain10x64.sys
```

Solving the high number of individuals and variables in the data

First of all, we did a correlation matrix in order to see if there were variables with a low correlation with the objective variable.



As we can see in the picture, in the correlation matrix appears the number of the correlation and it is not readable, so we print the correlation between the objective variable (Credit_Score) and the rest of the variables.



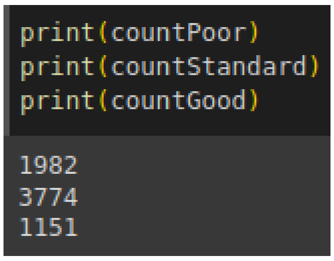
```
Age                          0.163389
Occupation                  -0.009202
Monthly_Inhand_Salary        0.208351
Num_Bank_Accounts           -0.375450
Num_Credit_Card             -0.396127
Interest_Rate               -0.482415
Num_of_Loan                 -0.348116
Delay_from_due_date         -0.427018
Num_of_Delayed_Payment      -0.360087
Num_Credit_Inquiries        -0.434796
Credit_Utilization_Ratio     0.037941
Credit_History_Age           0.381511
Payment_of_Min_Amount       -0.289043
Total_EMI_per_month         -0.099545
Amount_invested_monthly      0.143763
Payment_Behaviour           -0.113797
Monthly_Balance              0.191985
Credit_Score                 1.000000
Name: Credit_Score, dtype: float64
```

As we can see, there are some variables with a low correlation like: Occupation, Credit_Utilization_Ratio and others. But if we remove them, we will continue with the same problem, so we kept them and tried to do another thing

To solve it, we took 15 % of the original data without null values because we thinked it would be better to work with the original data and not with the KNN imputation because the KNN will have artificial values and we will have the same individuals.

The 15% of the data was taken with a sample that takes the data randomly with the same probability of each individual, so it should return the data with more or less the same distribution that was with all the data.

Unbalanced data

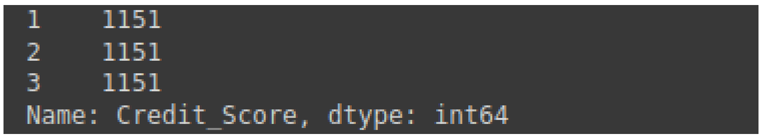


```
print(countPoor)
print(countStandard)
print(countGood)
```

```
1982
3774
1151
```

As we can see in the left picture, we have unbalanced data.

We solve it by using RandomUnderSampler. This method balances the data by removing some individuals. So now, we have 1151 samples in each label of the objective variable (Credit_Score)



```
1    1151
2    1151
3    1151
Name: Credit_Score, dtype: int64
```

After that, we did al the preprocess and export the data

Machine learning methods

In this section, we will proceed to summarize the decisions taken for the design of the different machine learning models we have carried out, together with a sample of their results and the conclusions drawn from each one.

Naïve Bayes

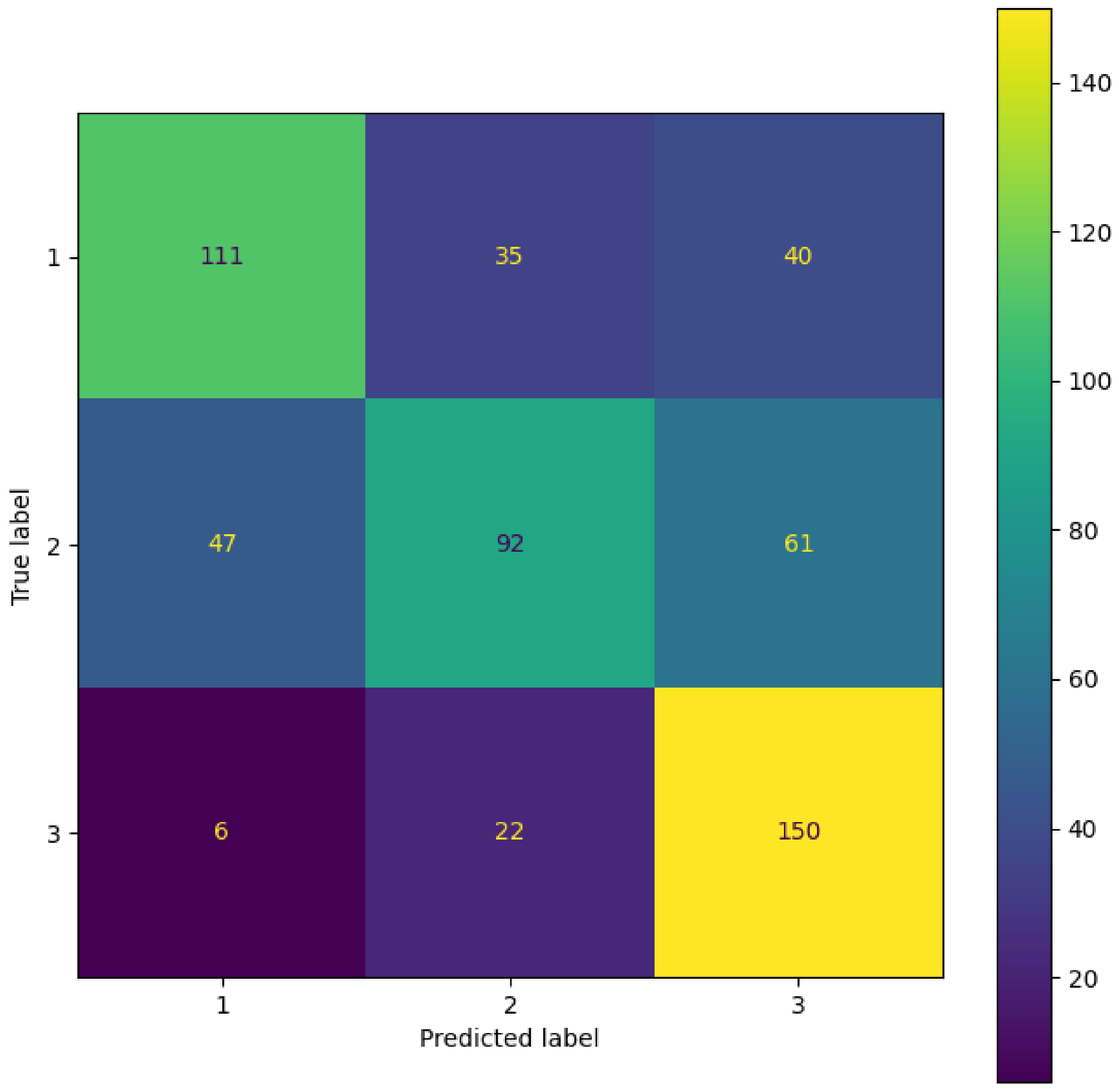
The first step in the development of our (and every) Naïve Bayes models is to normalize the data, as not doing so reduces the accuracy of our models.

Once we have normalized the data we perform a k-fold cross validation with 10 folds and calculate the mean and variance of their scores, giving us a result with a mean accuracy of 62.39% and a negligible Variance, which means that the results given by our model are quite poor.

Then, we trained the model with our training set and validated its performance with the test set, obtaining the following results:

	Precision	Recall	F1-Score	Support
Good	0.60	0.68	0.63	164
Standard	0.46	0.62	0.53	149
Poor	0.84	0.60	0.70	251
Accuracy			0.63	564
Macro Avg	0.63	0.63	0.62	564
Weighted Avg	0.67	0.63	0.63	564

The predictions of our model can be seen in the following confusion matrix:



As we can see, the results obtained with this model are quite poor, with an F1 score of 0.63 and a large number of erroneous results. In the case of the dataset used, the hypothesis of independence of attributes is not satisfied, as there are variables with a strong correlation between them.

KNN

As in all models, we start by performing a k fold cross validation with 10 partitions calculating the mean accuracy and variance, obtaining a mean accuracy of 66.49% and a negligible variance. So we can see that this model can give us slightly better results than Naïve Bayes.

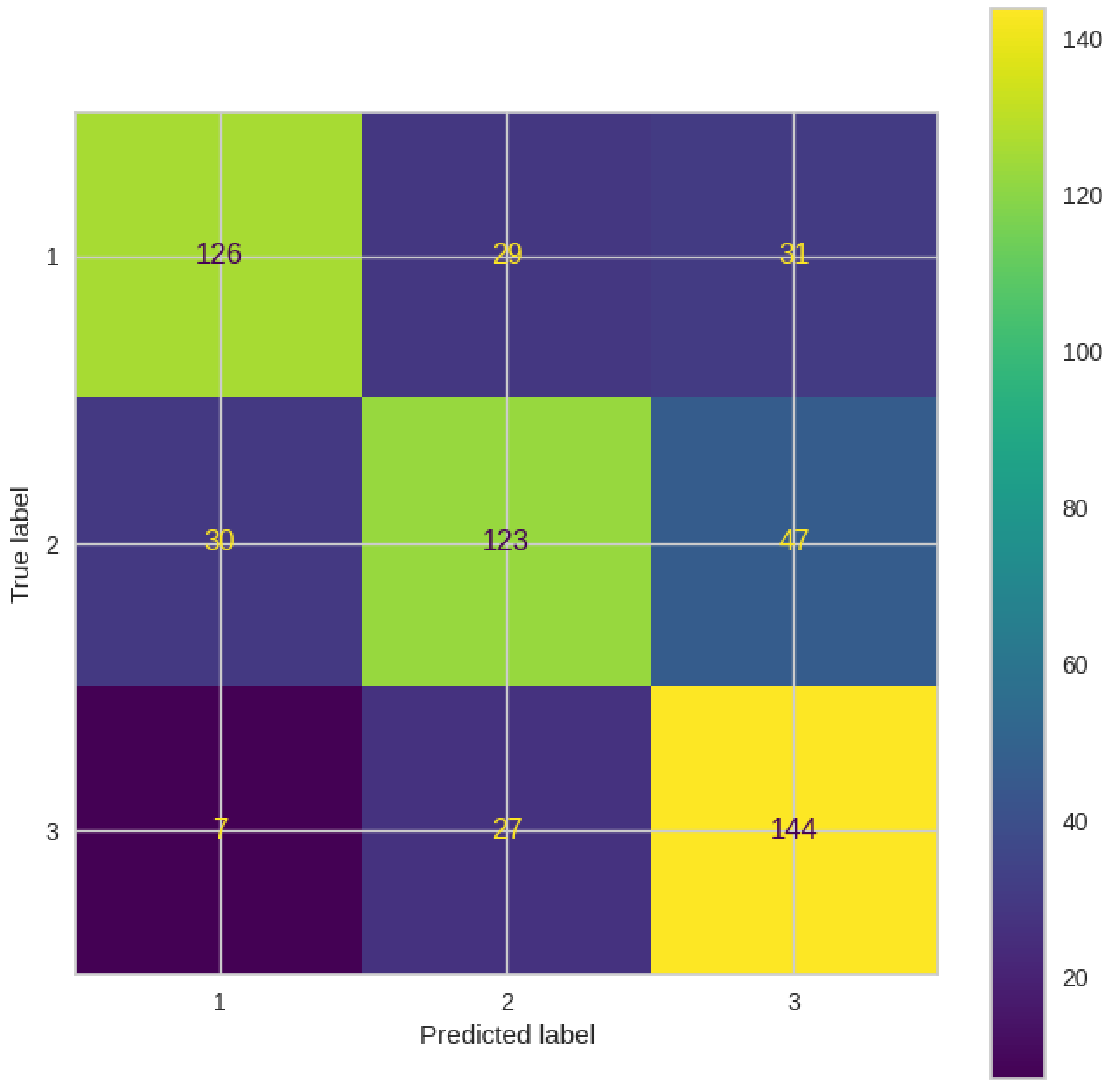
Then, we have trained the model by running a GridSearch with different values for our parameters, so that we get the best possible parameters. The results of this search show that the parameters that give us the best results are:

- Leaf Size: 20
- Metric: L1
- n_neighbors: 5
- Weights: distance

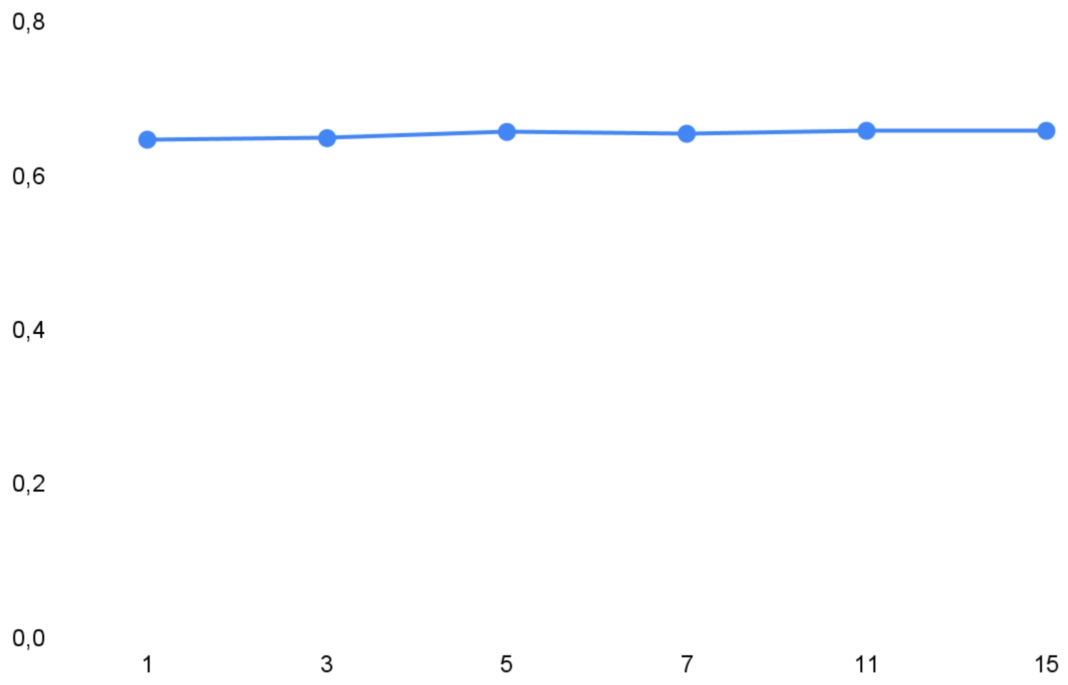
Once the model has been trained with the most optimal parameters for our dataset, we proceed to check its performance by comparing its predictions with the test set, obtaining the following results:

	Precision	Recall	F1-Score	Support
Good	0.68	0.77	0.72	163
Standard	0.61	0.69	0.65	179
Poor	0.81	0.65	0.72	222
Accuracy			0.70	564
Macro Avg	0.70	0.70	0.70	564
Weighted Avg	0.71	0.70	0.70	564

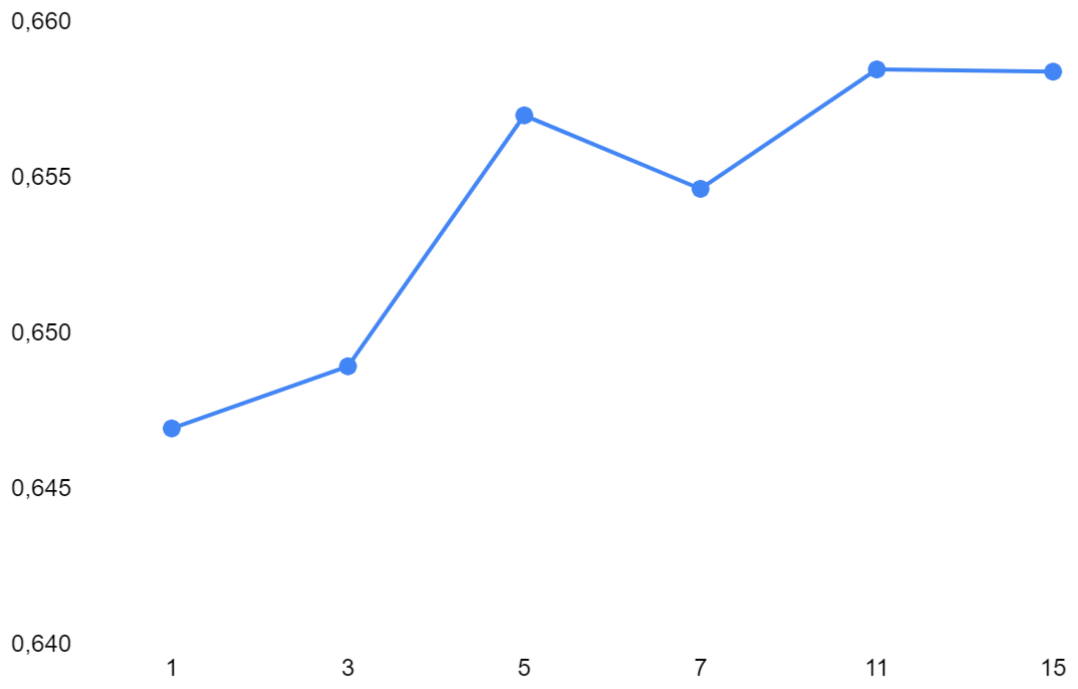
The predictions of our model can be seen in the following confusion matrix:



We will now visualize a comparison of the results obtained according to the k-value for our model.



As we can see, the difference is minuscule, let's scale the graph to better appreciate this difference.



Decision Trees

In the case of the decision tree model, when performing the 10-fold cross validation, we obtain an accuracy value of 60.65% and a variance of 0.17%, so we can say that it is a very slightly worse result than the rest seen so far.

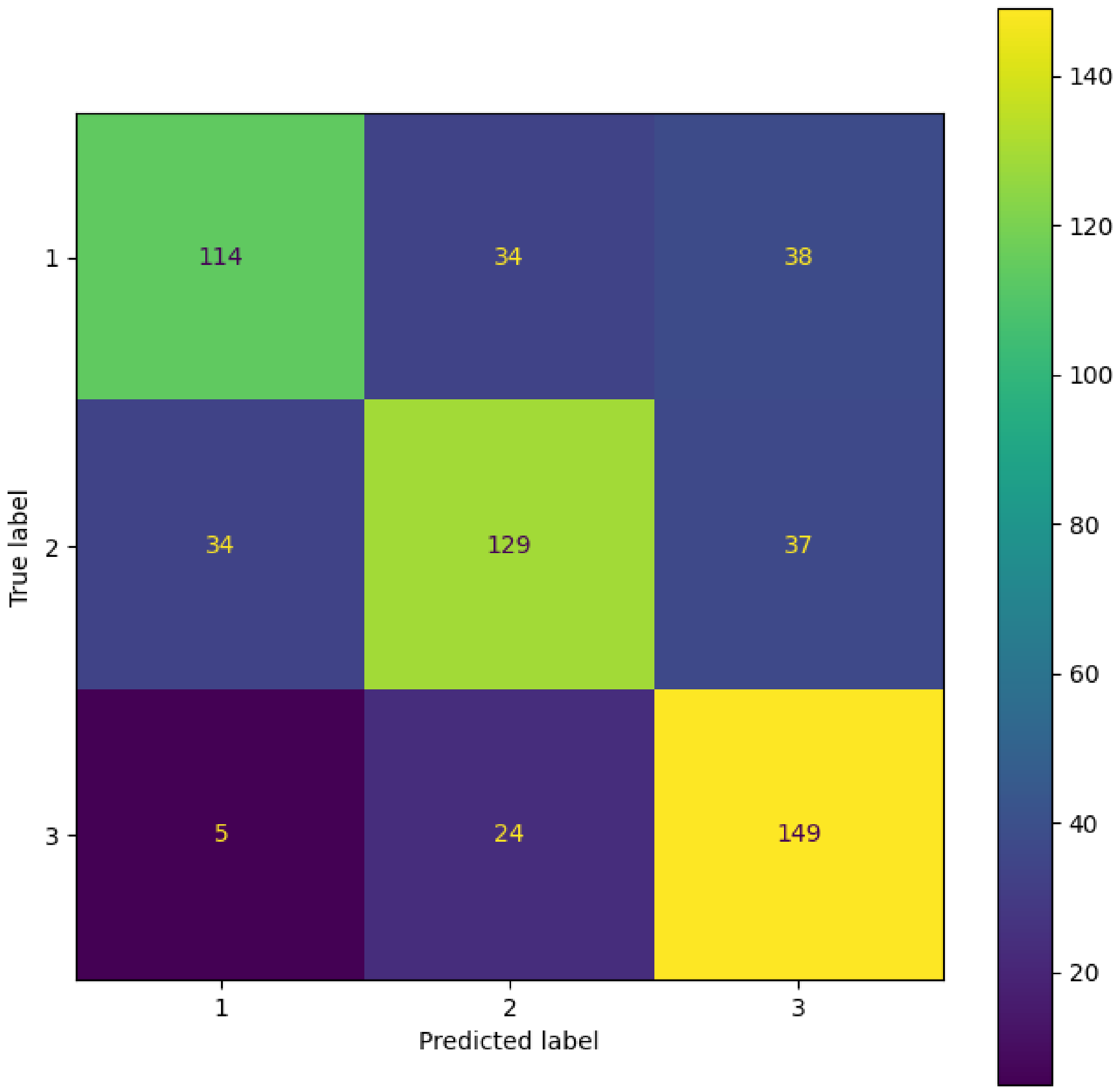
Then, we proceed to do a Bayesian search to find the best parameters to train our model with the dataset we have. After the search, we see that the optimal combination of parameters is:

- Criterion: Entropy
- Max Depth: 7
- Max Leaf Nodes: 10
- Min Samples Leaf: 5
- Splitter: Random

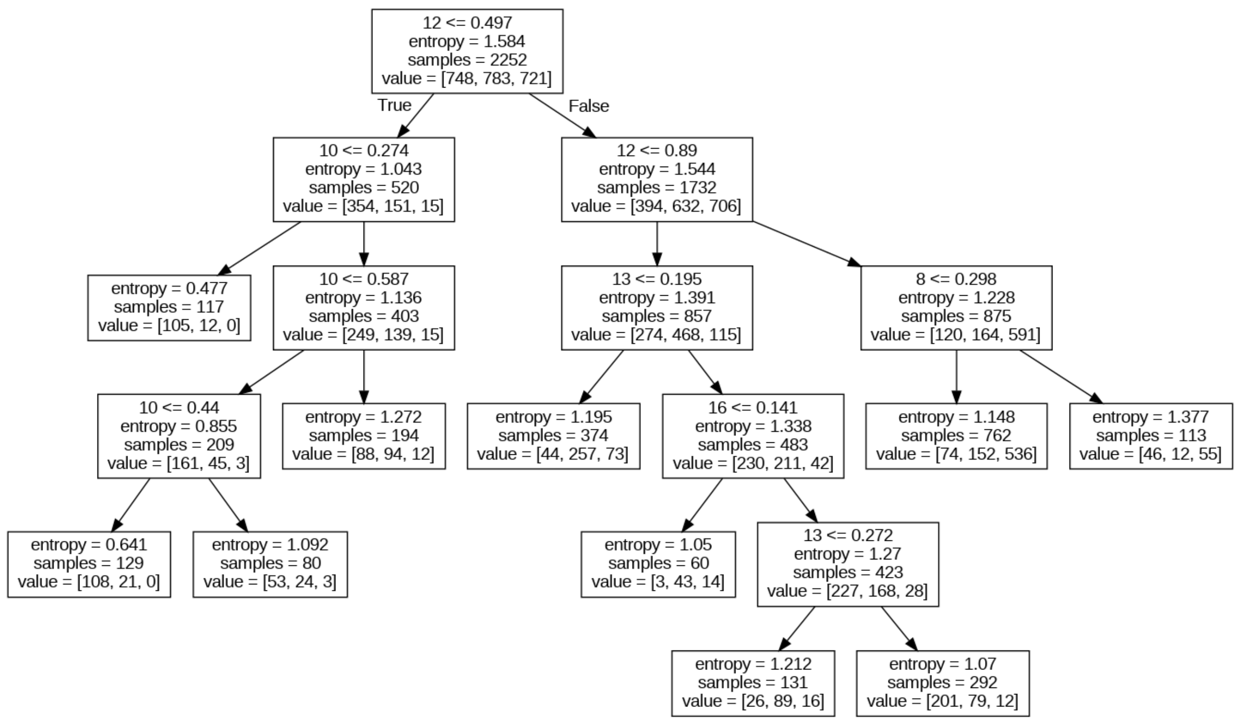
Once the model has been trained with the optimal combination of parameters, we proceed to validate the results with the test set, obtaining the following results:

	Precision	Recall	F1-Score	Support
Good	0.61	0.75	0.67	153
Standard	0.65	0.69	0.67	187
Poor	0.84	0.67	0.74	224
Accuracy			0.70	564
Macro Avg	0.70	0.70	0.69	564
Weighted Avg	0.71	0.70	0.70	564

The predictions of our model can be seen in the following confusion matrix:



Finally, the decision tree is printed to better understand how it makes decisions to classify individuals.



Support Vector Machine

In the case of the SVM, we see that, after the 10-fold cross validation, this model is the one with the best results so far, with a mean accuracy of 67.05% and a variance of 0.16%.

Then, we proceed to do a Bayesian search to find the best parameters to train our model with the dataset we have. After the search, we see that the optimal combination of parameters is:

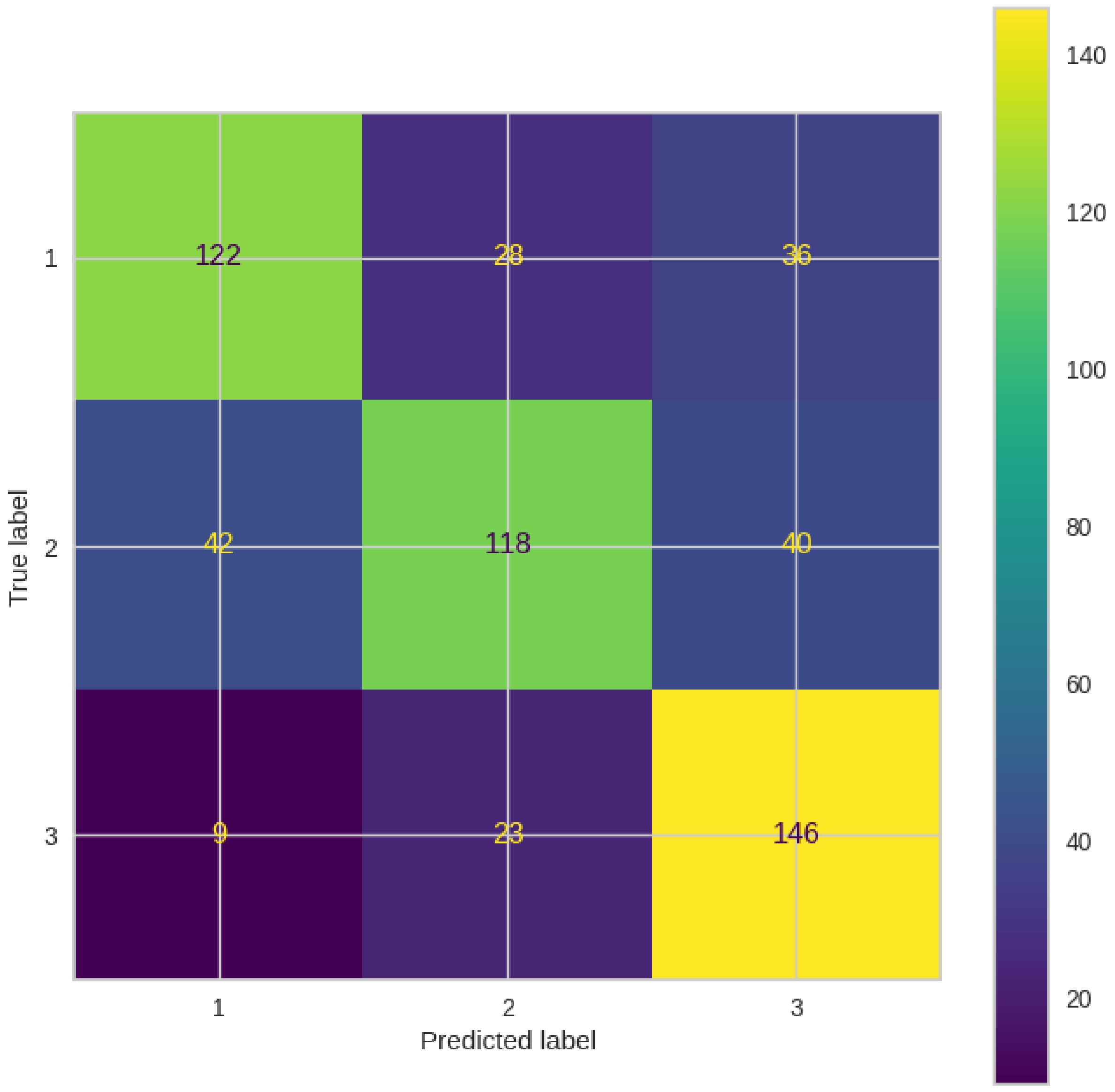
- C: 5.25
- Gamma: Scale

We have chosen to use the RBF kernel for our SVM, so that it can capture non-linear relationships in the data.

Once the model has been trained with the optimal combination of parameters, we proceed to validate the results with the test set, obtaining the following results:

	Precision	Recall	F1-Score	Support
Good	0.66	0.71	0.68	173
Standard	0.59	0.70	0.64	169
Poor	0.82	0.66	0.73	222
Accuracy			0.68	564
Macro Avg	0.69	0.69	0.68	564
Weighted Avg	0.70	0.68	0.69	564

The predictions of our model can be seen in the following confusion matrix:



Performance majority voting

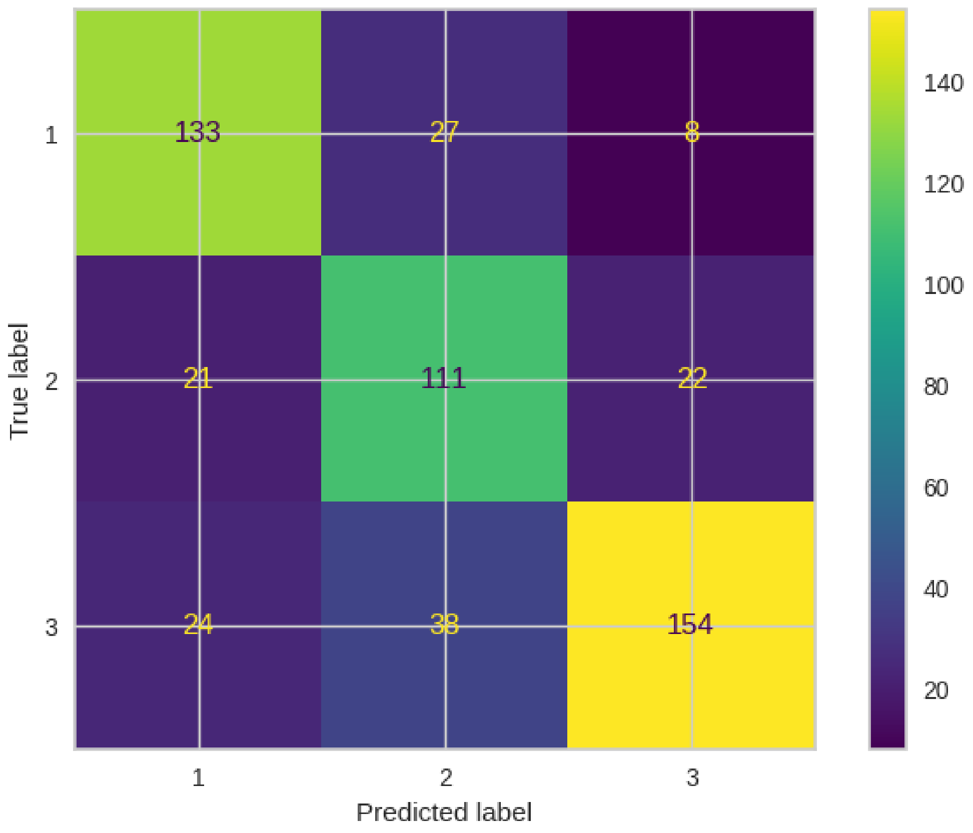
For the performance majority voting I first used the collection of base models we had such as: Naive Bayes, K-Nearest Neighbors (KNN), Decision Tree and Support Vector Machines (SVM). I used the predictions that resulted from their training and started by implementing a simple voting model with a ‘’loose’’ approach. The resulting prediction was the one that was the most present so basically equal to the mode.

I also looked and implemented a weighted majority voting model by assigning weight to predictions made by a model based on its accuracy and then multiplying the weight by the predictions of each model and then summing and rounding it to get a majority vote. However the accuracy of this algorithm was lower than with the simple majority voting so I decided to stick to that one.

The majority voting presents a mean accuracy of 73.98% which means the model has a decently good accuracy. Here are the results of this first implementation:

	Precision	Recall	F1-Score	Support
Good (1)	0.75	0.79	0.77	168
Standard (2)	0.63	0.72	0.67	154
Poor (3)	0.84	0.71	0.77	216
Accuracy			0.74	538
Macro Avg	0.74	0.74	0.74	538
Weighted Avg	0.75	0.74	0.74	538

The predictions of the majority voting model can be seen below in the confusion matrix:



Random Forest

The random forest model, after performing the 10-fold cross validation, presents a mean accuracy of 71.14% and a variance of 0.11%, so we can see that the results it will give us will be relatively good.

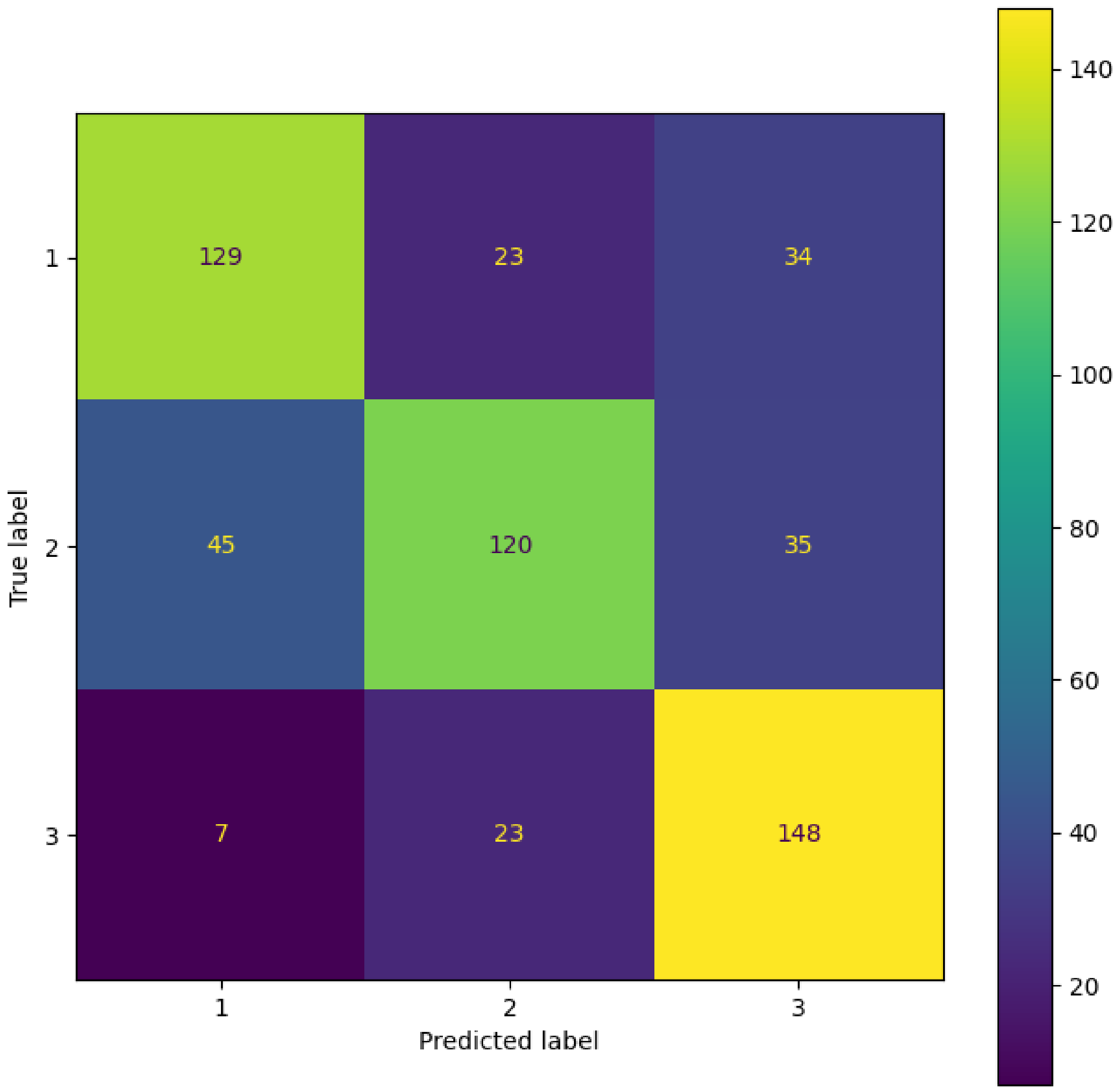
Once we have seen that the results to be received will be good enough, we perform a Bayesian search to find the optimal combination of parameters for our model, obtaining the following results:

- Criterion: Entropy
- Max Depth: None
- Min Samples Leaf: 2
- N Estimators: 100

Once we have discovered the best parameters for our model, we proceed to train it with these parameters. Once the model is trained, we validate its results with the test set, obtaining the following results:

	Precision	Recall	F1-Score	Support
Good	0.69	0.71	0.70	181
Standard	0.60	0.72	0.66	166
Poor	0.83	0.68	0.75	217
Accuracy			0.70	564
Macro Avg	0.71	0.71	0.70	564
Weighted Avg	0.72	0.70	0.71	564

The predictions of our model can be seen in the following confusion matrix:

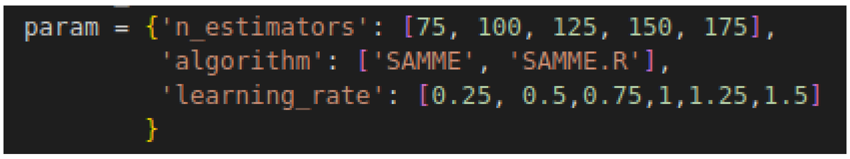


True label
Predicted label
1
2
3
129
23
34
45
120
35
7
23
148
20
40
60
80
100
120
140

Adaboost

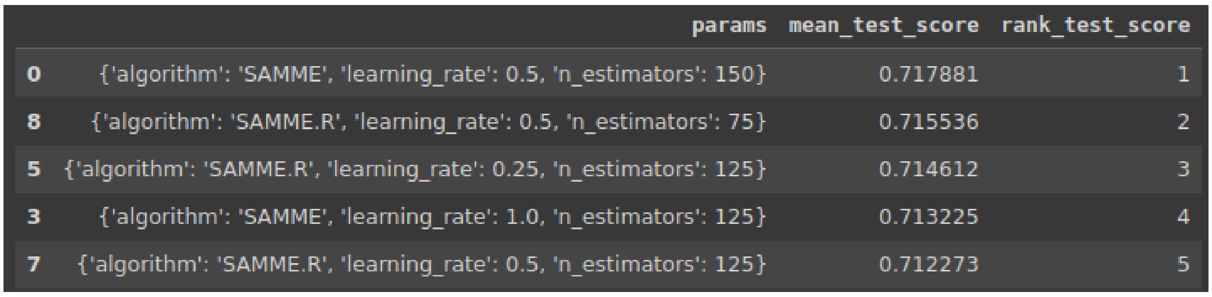
Initially, we do a normalization with the MinMaxScaler as all the models before. And then we tried to adjust the model to the data by searching for better hyperparameters. As we can see in the picture below, we are finding the three hyperparameters that are the next:

- n_estimators: The maximum number of estimators at which boosting is terminated. In case of perfect fit, the learning procedure is stopped early
- learning_rate: Weight applied to each classifier at each boosting iteration. A higher learning rate increases the contribution of each classifier.
- algorithm: If ‘SAMME.R’ then use the SAMME.R real boosting algorithm. estimator must support calculation of class probabilities. If ‘SAMME’ then use the SAMME discrete boosting algorithm. The SAMME.R algorithm typically converges faster than SAMME, achieving a lower test error with fewer boosting iterations.



```
param = {'n_estimators': [75, 100, 125, 150, 175],
         'algorithm': ['SAMME', 'SAMME.R'],
         'learning_rate': [0.25, 0.5,0.75,1,1.25,1.5]
        }
```

After that, we find the best hyperparameters by doing a BayesSearchCV:



	params	mean_test_score	rank_test_score
0	{'algorithm': 'SAMME', 'learning_rate': 0.5, 'n_estimators': 150}	0.717881	1
8	{'algorithm': 'SAMME.R', 'learning_rate': 0.5, 'n_estimators': 75}	0.715536	2
5	{'algorithm': 'SAMME.R', 'learning_rate': 0.25, 'n_estimators': 125}	0.714612	3
3	{'algorithm': 'SAMME', 'learning_rate': 1.0, 'n_estimators': 125}	0.713225	4
7	{'algorithm': 'SAMME.R', 'learning_rate': 0.5, 'n_estimators': 125}	0.712273	5

As we can see in the picture, the best hiperparameters are with the algorithm ‘SAMME’, with a learning_rate of 0.5 and with n_estimators = 150 because is the higher mean test score with 0.7179, but they are more or less the same score.

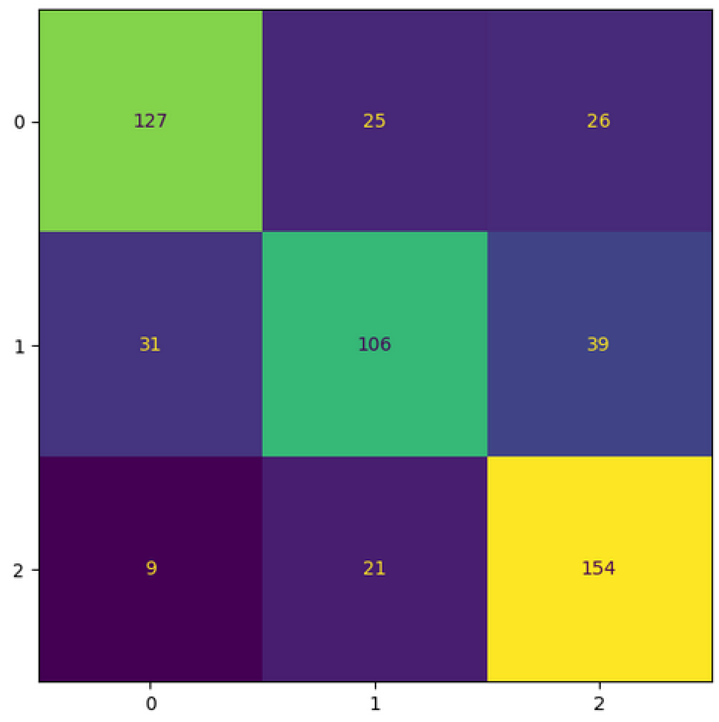
We think that the search of the hyperparameters are more or less adjusted, because in the parameters, the best result is not the maximum or the minimum value in the vector.

The Adaboost model, after performing the 10-fold cross validation, presents a mean accuracy of 71.79%, so we can see that the results that it gives us will be relatively good.

We analyze better the metrics by doing a table of the different classes and with different metrics as shown in the next table

	Precision	Recall	F1-Score	Support
Good	0.83	0.70	0.76	219
Standard	0.60	0.70	0.65	152
Poor	0.71	0.76	0.74	167
Accuracy			0.72	152
Macro Avg	0.72	0.72	0.72	538
Weighted Avg	0.73	0.72	0.72	538

And it have the next confusion matrix where 0 is Good, 1 is Standard and 2 is Poor, the left side is the real label and the below is the predicted label:



As we can see in the table and in the confusion matrix, the model is relatively good at classifying the data in general, but it predicts a bit better the label Good, then the Poor and finally the Standard label, because we have data that is in another label as we can see in the confusion matrix and we can quantify that, by looking at the different metrics like precision, recall or F1-Score or simply by looking at the confusion matrix which ones are in an incorrect label.

Bagging Classifier

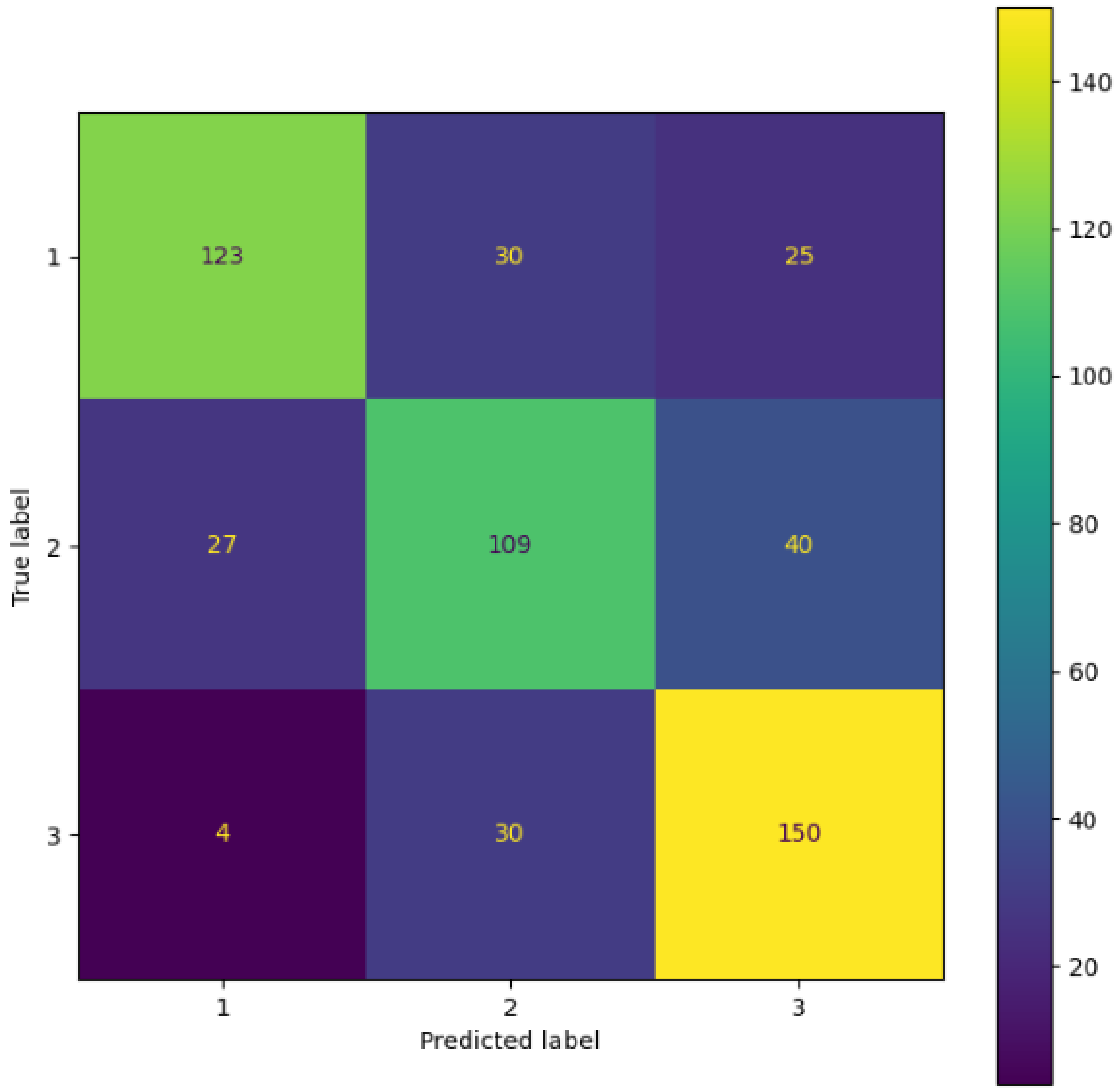
Once we have normalized the data we perform a k-fold cross validation with 10 folds and calculate the mean and variance of their scores, giving us a result with a mean accuracy of 72.67% and a variance of 0.001, which indicates stable performance.

Then, we trained the model with our training set and validated its performance with the test set, obtaining the following results:

	Precision	Recall	F1-Score	Support
Good	0.69	0.80	0.74	154
Standard	0.62	0.64	0.63	169
Poor	0.82	0.70	0.75	215
Accuracy			0.71	538
Macro Avg	071	0.71	0.71	538
Weighted Avg	0.72	0.71	0.71	538

The model achieved an overall accuracy of 71%, correctly classifying 71% of the instances. Precision varied across the classes, with class 1 having a precision of 69%, class 2 with 62%, and class 3 with 82%. Recall also varied, with class 1 having a recall of 80%, class 2 with 64%, and class 3 with 70%. The F1-scores were 0.74 for class 1, 0.63 for class 2, and 0.75 for class 3. These metrics indicate that the model achieved relatively balanced performance across the classes, with moderate levels of precision and recall.

The predictions of our model can be seen in the following confusion matrix:

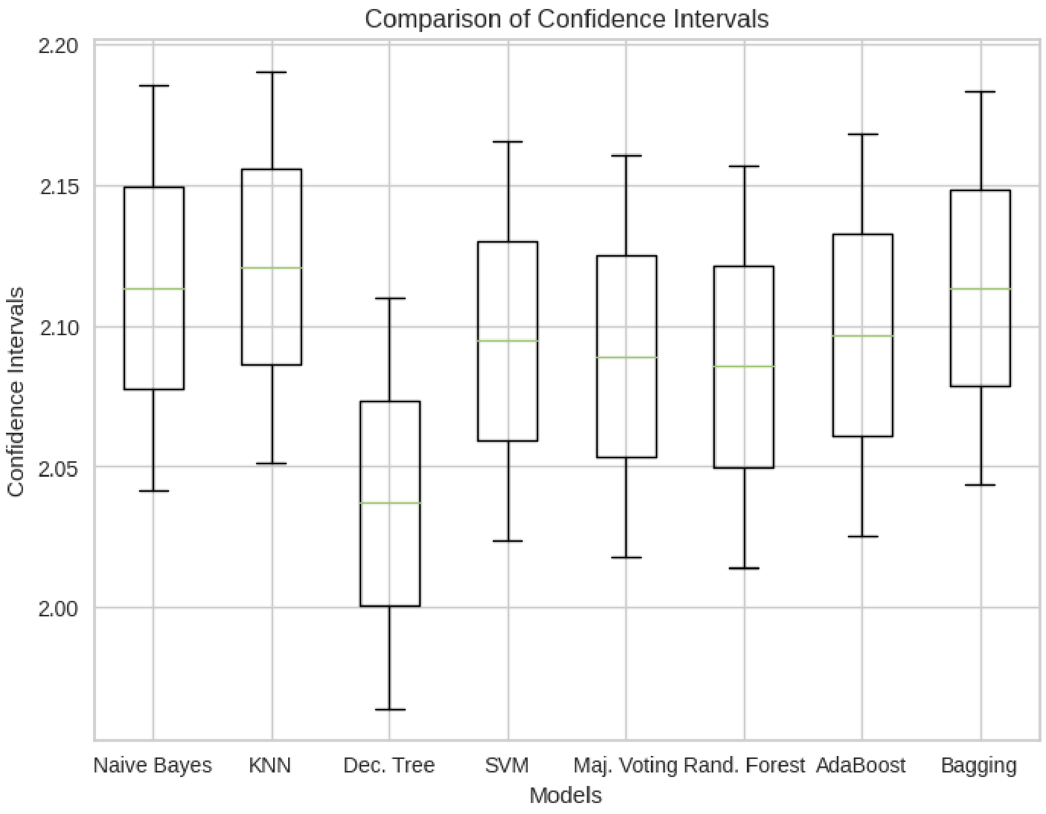


Comparisons and conclusions.

	Naive Bayes	KNN	Decision Tree	SVM	Majority Voting	Random Forest	AdaBoost	Bagging
Accuracy	62.39%	66.49%	60.65%	67.05%	73.98%	71.14%	71.79%	72.67%
Variance	Negli.	Negli.	0.17%	0.16%		0.11%	Negli.	Negli.

So as we can see in the comparative table, the base models all have less accuracy then meta-learning algorithms that allow us to get into the 70%+ accuracies. This makes sense because it is why those meta-learning algorithms were developed/invented. By using multiple base models to converge to a final prediction, the accuracy usually increases. We can also look at the intervals of confidence below:

Models	Confidence Intervals
Naive Bayes	(2.0416, 2.1852)
KNN	(2.0514, 2.1903)
Decision Tree	(1.9643, 2.1100)
SVM	(2.02410, 2.1655)
Majority Voting	(2.0180, 2.1604)
Random Forest	(2.0144, 2.1566)
AdaBoost	(2.0253, 2.1680)
Bagging	(2.0438, 2.1829)



The above confidence intervals were calculated with an alpha of 0.05. All models seem to have a similar confidence interval except the Decision Tree model which seems to be an outlier.

To conclude, we can establish that all meta-learning methods are good choices for final models, but because of the nature of the majority voting algorithm, we could add the Random Forest, AdaBoost and Bagging models as base models in the majority voting algorithm which might bring an even better accuracy. We therefore recommend to use the majority voting model from which to take the final results.